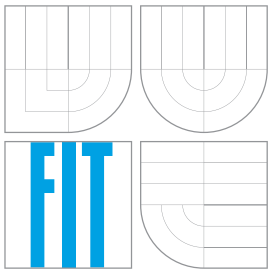
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

VYTVOŘENÍ MODELU PROCESORU POWERPC

MODELLING OF POWERPC PROCESSOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

HYNEK BLAHA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MASAŘÍK KAREL, Ph.D.

BRNO 2013

Abstrakt

Architektury procesorů jsou čím dál více složitější, proto je kladen velký důraz na automatizaci jejich návrhů. Tato bakalářská práce popisuje návrh procesoru PowerPC v jazyce pro popis architektur Codal. Funkčnost a výkonnost výsledného modelu byla ověřena testy poskytnutými výzkumnou skupinou Lissom a srovnána se současným konkurentem.

Abstract

Processor architectures are becoming increasingly complex, so great emphasis is put on the automation of their desings. This bachelor thesis describes the design of the PowerPC processor architecture in Codal language. The model is created according to avaliable documentation. The functionality and efficiency of the model was verified by tests provided by research group Lissom and compared to current competitor.

Klíčová slova

jazyk pro popis architektury, instrukční sada, toolchain, procesor, PowerPC, CISC, RISC, CodAL, Codasip, modelování, simulace

Keywords

architecture description language, instruction set, toolchain, processor, PowerPC, CISC, RISC, CodAL, Codasip, modelling, simulation

Citace

Hynek Blaha: Vytvoření modelu procesoru PowerPC, bakalářská práce, Brno, FIT VUT v Brně, 2013

Vytvoření modelu procesoru PowerPC

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Karla Masaříka, PhD. Další informace mi poskytla výzkumná skupina Lissom. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Hynek Blaha
13. května 2013

Poděkování

Děkuji vedoucímu mé práce Ing. Karlu Masaříkovi, Phd., a Ing. Adamu Husárovi za odborné rady a ochotnou pomoc.

© Hynek Blaha, 2013.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

Kapitola 1

Úvod

Tato bakalářská práce popisuje návrh procesoru PowerPC na úrovni instrukční sady v jazyce pro popis architektur CodAL.

Motivací pro tvorbu modelu procesoru je seznámení se s Codasip Studiem sloužícím k rychlému prototypování architektur a následné zapojení do jeho vývoje. V ideálním případě bude můj model následně komerčně využit.

Na začátku práce se snažím nastínit základní požadavky kladené na kvalitu vestavěných zařízení a situaci na jejich trhu 2. V následující kapitole 3 shrnuji poznatky o typech procesorů a klasifikuji je dle různých kritérií.

Kapitola 4 obsahuje stručné poznatky o specializovaných jazycích pro popis hardware a počítačových architektur. Čtenář bude seznámen s výhodami a nevýhodami těchto jazyků při návrhu procesoru.

Model procesoru PowerPC byl popsán s využitím jazyka pro popis architektur CodAL 5 a generován s pomocí Codasip Frameworku. Jeho vlastnostem a kvalitám je věnována kapitola 6.

Procesor PowerPC byl výkonným srdcem počítačů Macintosh firmy Apple až do roku 2006, kdy byl nahrazeny procesory Intel. Rodina procesorů PowerPC díky své flexibilitě a škálovatelnosti pokrývá široký okruh systémů od úsporných vestavěných zařízení až po vysoce výkonné sálové počítače pro vědecké výpočty a zpracování grafiky. Více o tomto procesoru přináší kapitola 7.

Kapitola 8 je ryze praktická a obsahuje postup a způsob modelování procesoru PowerPC. Popisuji v ní komplikace, se kterými jsem se v průběhu modelování setkal, a způsoby, jak jsem je vyřešil.

V poslední kapitole 9 srovnávám výkonnost vygenerovaného simulátoru se současným, vysoce optimalizovaným konkurentem PSIM, který se díky svým kvalitám stal součástí GNU PowerPC Debuggeru. Oba simulátory byly srovnány třemi testy vybranými ze sedmi set prvkové sady, která mi byla poskytnuta výzkumnou skupinou Lissom za účelem odladění modelu.

Kapitola 2

Teorie

Informace v této kapitole byly čerpány z [11, 10].

Více než 90 % vyrobených procesorů je používáno ve vestavěných systémech. Vestavěným systémem rozumíme systém, jehož celý řídicí počítač je zabudován do zařízení, které ovládá. Na rozdíl od univerzálních počítačů, po kterých je vyžadována široká funkčnost, jsou vestavěná zařízení tvořena jednoúčelově. Vzhledem k tomu mají návrháři možnost cílené optimalizace, která se promítne do snížené ceny finálního výrobku. V dnešní vyspělé době jsme obklopeni elektronikou doslova na každém kroku. Typickým reprezentantem vestavěného zařízení je například telefon, bankomat, alarm pro zabezpečení objektů či mikrovlnná trouba pro ohřev jídel.

Od vestavěných zařízeních požadujeme nízkou cenu (sériová výroba ve velkých objemech), malou spotřebu (měřič rychlosti u silnice napájený solárními články nemůže po dvou zatažených dnech přestat fungovat), malé rozměry (ušetřená užitková plocha na čipu). Zároveň jsou často kladeny protichůdné požadavky na fungování.

Vestavěné zařízení má mít vysoký výkon (digitální televizní přijímač musí každou sekundu zpracovat velké množství dat), mohou být kladeny požadavky na fungování v reálném čase (protiraketový systém musí chránit okamžitě), na bezpečnost (nesmí dovolit riziko v jaderných elektrárnách) či na spolehlivost (robot sbírající vzorky půdy z Marsu se nesmí rozbít).

Výsledná kritéria pro hodnocení kvality vestavěných systémů mohou být:

- Cena za kus - Určuje konečnou cenu produktu, tím jeho dostupnost masám lidí, objem prodeje a zisk firmy.
- Kvalita návrhu - Souvisí s ní schopnost plnit kladené požadavky na produkt.
- Úspěch na trhu - Často určen poměrem cena/výkon, ale i inovativním řešením oproti konkurenci.

Spolehlivost vestavěných zařízení můžeme určit podle následujících kritérií:

- Systém je pro opravu nedostupný - zařízení ve vesmíru, podmořské kabely, sériově vyráběná spotřební elektronika
- Zastavení systému způsobuje finanční ztráty - řízení továren, ovládání mostů, spravování bezhotovostních plateb

- Systém nesmí běžet v nekorektním stavu - automatické brzdy, lékařské přístroje, detekce hrozeb v jaderné elektrárně
- Požadavky na neustálý chod systému - navigace letadel, obraný protiraketový systém

Obecně platí, že se ve vestavěných systémech pro zpracování složitých úloh používají CISC procesory (Intel Atom, M68K). Nízkospotřební či kritické aplikace jsou častěji vykonávány v RISC procesorech, jako je například PowerPC. V následující kapitole bude čtenář seznámen s typickými rysy těchto architektur.

Kapitola 3

Klasifikace procesorů

Informace v této kapitole byly čerpány z [5, 10, 9].

Tato kapitola pojednává o klasifikaci procesorů na základě charakteristik instrukční sady a cílového zaměření. V současné době se ryzí CISC nebo RISC architektury téměř nevyskytují. Vždy jde o jakýsi kompromis mezi oběma přístupy a výsledný typ procesoru často závisí na marketingu firmy.

3.1 Podle instrukční sady

3.1.1 CISC - Complex Instruction Set Computer

Complex Instruction Set Computer neboli CISC označuje skupinu procesorů, které se vyznačují podobným návrhem sady strojových instrukcí. Ty formou mikroprogramů v PROM pokrývají velmi široký okruh funkcí, které by jinak bylo možné popsat skupinou jednodušších instrukcí.

Snahou při tvorbě komplexních instrukcí byla lepší podpora konstrukcí vysokoúrovňových jazyků. Díky na míru konstruovaným instrukcím se zmenšil počet přístupů do pomalých pamětí a spolu s tím se zvýšil výkon. Výhody sémanticky košatých instrukcí byly využity v cache procesorů, protože jejich kód je kratší a snadněji se do nich vejde.

Díky velké specifičnosti bylo časté, že část sady komplexních strojových instrukcí nebyla téměř využívána. Procesor byl řízen instrukcemi různých délek s mnoha parametry, proto musely být používány složité dekodéry. Doby provádění operací trvaly různý počet taktů hodinového signálu, závisely na komplikovanosti instrukce a místě uložení operandů.

Extenzivní rozvoj instrukčních sad, při kterém se nová instrukce přidala k již hotovým, vedl ke stavu, kdy rozšiřování paměti ROM, kde byly uloženy mikroprogramy instrukcí, bylo dále neúnosné.

		%
LOAD	čtení z paměti	26.6
STORE	zápis do paměti	15.6
Jcond	podmíněný skok	10.0
LOADA	načtení adresy	7.0
SUB	odčítání	5.8
CALL	volání podprogramu	5.3
SLL	bitový posun vlevo	3.6
IC	vložení znaku	3.2

		%
LOAD	čtení z paměti	27.3
Jcond	podmíněný skok	13.7
STORE	zápis do paměti	9.8
CMP	porovnání	6.2
LOADA	načtení adresy	6.1
SUB	odčítání	4.5
IC	vložení znaku	4.1
ADD	sčítání	3.7

Obrázek 3.1: Statistická a dynamická četnost instrukcí procesoru IBM System/360 Zdroj[9]

V 70. letech minulého století byly prováděny výzkumy četnosti instrukcí v programech. Zjistilo se, že je programátory i kompilátory využíván jen úzký okruh sady strojových instrukcí (viz obrázek 3.1).

Následně byly vzneseny návrhy, zda by tyto nepoužívané instrukce neměly být z instrukční sady odstraněny, a tím zjednodušen řadič procesoru.

3.1.2 RISC - Reduced Instruction Set Computer

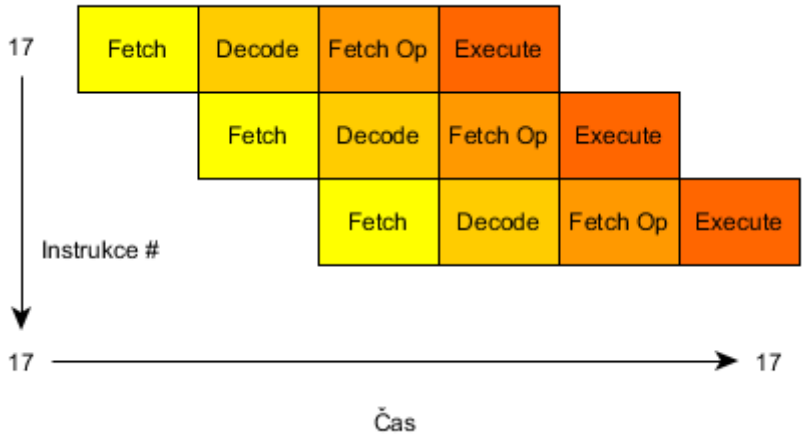
Základním znakem procesorů typu RISC je omezená instrukční sada s pevnou délkou instrukce, jednotným formátem a malým počtem možností adresování. V závislosti na zjednodušení instrukcí přestává být potřebný složitý řadič. I datové cesty se zjednodušují, a proto může procesor pracovat na vyšší frekvenci.

S pamětí nelze provádět jiné operace než instrukce typu LOAD/STORE, výpočty probíhají nad rychlejšími registry, kterých je větší množství než u CISC procesorů. Pro zkrácení času přístupu do paměti tyto architektury hojně využívají pamětí cache, do kterých v pravidelných intervalech načítají více instrukcí. Zjednodušení technického vybavení je kompenzováno optimalizacemi v kompilátoru.

Všechny tyto vlastnosti mají za cíl vytvořit jednoduchý, ale zároveň efektivní celek. Takto pojaté procesory přinášejí výhody ve formě jednoduššího návrhu, se kterým jde ruku v ruce zkracování doby vývoje.

Jednotlivé instrukce jsou implementovány logickými obvody, rychlejší alternativou k mikroprogramům CISC architektury.

V každém strojovém cyklu je díky zřetězenému zpracování (viz obrázek 3.2) dokončena alespoň jedna instrukce. To pracuje na principu vykonávání elementárních operací, které na sebe navazují a vzájemně se doplňují v souhlase s algoritmem požadovaného zpracování.



Obrázek 3.2: Demonstrace zřetězeného zpracování instrukcí

Pevná délka instrukcí podporuje snazší načítání z paměti, a tím zajišťuje pravidelné plnění fronty. K identifikaci instrukcí jednotného formátu navíc postačí jednodušší dekodér. Další výhodou je nedestruktivní zpracování operandů díky práci s trojadresným kódem, a tak jsou eliminovány nepříjemnosti s opakovaným výpočtem původních hodnot.

RISC architektury, které paralelizací částí procesoru dosahují lepších výsledků (více než jedna zpracovaná instrukce za jednotku doby), se nazývají superskalární.

3.2 Podle zaměření

3.2.1 CPU - Univerzální procesory

Univerzální procesory jsou velice flexibilní, poskytují velký okruh softwarově implementovaných funkcí, protože není dopředu známé, které programy budou zpracovávat. Lze je tedy využít téměř ve všech aplikacích.

Daní za vysokou flexibilitu je jejich nižší výkon. Výslednou funkčnost určuje software, který na daném procesoru běží.

Tyto procesory se nejčastěji využívají v osobních počítačích.

3.2.2 ASIP - Aplikačně specifické procesory

Procesory s aplikačně specifickou instrukční sadou jsou vytvářeny na míru pro konkrétní účel a díky tomu mohou být vysoce optimalizované.

Jejich obvodově řešená instrukční sada je oproti univerzálním procesorům značně užší, ale výkonnější. Návrh aplikačně specifických procesorů je dlouhodobý a nákladný. Ve výsledné architektuře jsou použity pouze komponenty požadované pro chod systému. Tyto procesory mohou mít konfigurovatelnou instrukční sadu. Základní ISA je napevno určena logickými obvody, zbylé instrukce lze naprogramovat syntézou programovatelného hradlového pole (FPGA).

Počáteční vysoké náklady vynaložené na návrh procesorů s aplikačně specifickou instrukční sadou se vrací ve formě výsledného levného výrobku, který je produkován ve velkém množství a přesně splňuje kladené požadavky. Těmi mohou být často protichůdné

nároky na vysoký výkon, garantovaný čas odpovědi kombinované se snahou o co nejmenší spotřebu.

Integrací vzájemně propojených hardwarových komponent (procesor, paměť a jiné) vznikají takzvané systémy na jednom čipu (SoC - System on Chip), které jsou často využívány k implementaci procesorů ASIP. Tyto platformy zrychlují návrh systému, neboť je lze programovat ve vyšších programovacích jazycích.

Následující kapitola popisuje způsoby návrhu procesorů a jazykové prostředky, které jsou v průběhu vývoje používány za účelem automatizace.

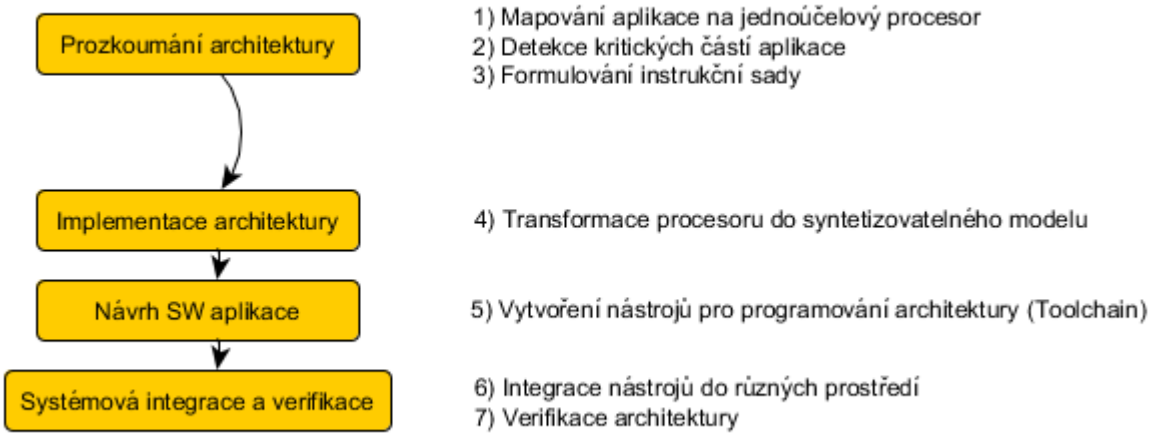
Kapitola 4

Jazyky pro popis procesorů

Informace v této kapitole byly čerpány z [8].

S pravidelným zvyšováním počtů tranzistorů v integrovaných obvodech, kde se každé dva roky jejich počet přibližně zdvojnásobí[1], je zvyšována složitost výsledného výrobku.

Tradiční návrh, při kterém skupina profesionálů navrhuje procesor s nízkou úrovní automatizace, se zachoval dodnes. Vzhledem ke komplexnosti návrhu spolu musí spolupracovat často oddělené týmy vývojářů, a protože jde o velice pracný proces, výsledkem často nebývá nejlepší možné řešení. Pokud ano, trvá to dlouhý čas.



Obrázek 4.1: Tradiční vývoj architektury

[1] http://kisk.phil.muni.cz/wiki/Moorův zákon

4.1 HDL - Jazyky pro popis hardware

Druhým možným přístupem je automatizovaný návrh za pomoci jazyků k tomu určených.

Jazyky pro popis hardware (HDL) na rozdíl od jiných v sobě obsahují prvek časování ve formě zpracování hran hodinového signálu. Proto mohou být využívány pro návrh a simulace hardware.

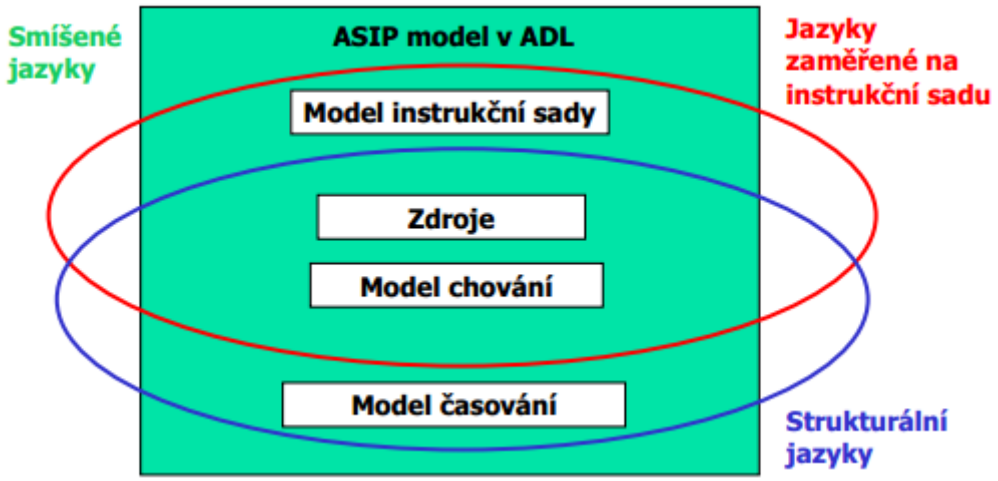
Procesory a jejich neustále se zvyšující složitost se jimi ovšem popisují špatně. Zároveň HDL neumožňují generování programovací (kompilátor, assembler, linker atd.) a simulační (debugger, profiler) toolchain. Proto by byl návrh procesorů značně komplikovaný a pracný.

Do skupiny jazyků pro popis hardware patří například VHDL nebo Verilog.

4.2 ADL - Jazyky pro popis architektury

Výše zmíněné problémy překonávají jazyky pro popis architektury (ADL).

Cenou za vyšší zaměření, než mají HDL, je zanedbání určitých parametrů popisovaného procesoru, jakými je jeho rozloha či taktování. Jazyky pro popis architektury mohou podporovat možnosti kontroly konzistence, výkonu a jednoznačnosti formálního zápisu. Díky tomu dokáží některé z nich automaticky vytvářet z formálního popisu nástroje potřebné pro další fáze vývoje (assembler, disassembler, simulátor atd.).



Obrázek 4.2: Zaměření jazyků ADL. Převzato z [8]

Jazyky zaměřené na instrukční sadu jsou nejčastěji využívány s cílem vyvinout přenositelný překladač vyšších programovacích jazyků. Soustředí se na modelování instrukční sady pomocí atributované gramatiky a časování uvnitř architektury.

Oproti tomu se strukturální jazyky soustředí na komponenty a jejich zapojení. Kvůli nižší úrovni abstrakce procesoru jsou generované nástroje pomalejší.

Smíšené jazyky tyto dva přístupy kombinují. Do této skupiny spadá jazyk pro popis architektur CodAL, který jsem využil k vytvoření modelu procesoru PowerPC, je věnována následující kapitola.

Kapitola 5

Jazyk CodAL

Informace v této kapitole vycházejí z [6].

Jazyk CodAL byl vyvinut za účelem rychlého prototypování procesorů s aplikačně specifickou instrukční sadou výzkumnou skupinou Lissom na Fakultě informačních technologií VUT v Brně a spadá do kategorie smíšených jazyků pro popis architektury.

To znamená, že současně povoluje popis architektury i instrukční sady. Je inspirován jazykem LISA a splňuje současné potřeby kladené na souběžný vývoj hardwaru a softwaru tím, že je schopen automaticky generovat nástroje potřebné pro programování a simulaci ze zadaného popisu architektury.

Definice procesoru zapsaná v jazyku CodAL je po zkontrolování přeložena do XML formátu, který je následně čten a zpracováván generátory nástrojů (assembler, disassembler, překladač jazyka C, linker, simulátor atd.). Jelikož jsou tyto nástroje vytvářeny velice rychle, může být úpravami v popisu architektury a instrukční sady vyzkoušeno více variant výsledného produktu.

Model procesoru je definován jednou ze čtyř úrovní složitosti popisu:

- Instrukční model
- Časový model
- Model chování
- Strukturální model

Ačkoliv lze modely popisovat s různou úrovní detailu, základní struktura souboru je stejná napříč všemi typy. Aby bylo možné model úspěšně přeložit, musí obsahovat popis zdrojů, instrukcí a událostí.

5.1 Popis zdrojů

V této části jsou definovány všechny hardwarové zdroje, jako jsou registry, paměť, sběrnice a jiné.

U popsaných prvků může být modelováno jejich propojení s jinými, například mapování paměti do adresového prostoru. Je vyžadováno, aby byl čítač instrukcí (PC) modelován jako speciální prvek.

5.2 Popis instrukcí a událostí

Druhá povinná část obsahuje popis instrukcí a reakcí na určité události, jako je jejich zpracování. Je vyžadováno, aby byly popsány události Reset, Halt, Main a alespoň jedna instrukce.

- Reset - definuje akce, které mají být vykonány při resetu procesoru
- Halt - definuje akce, které mají být vykonány na konci simulace (výpis registrů atd.)
- Main - definuje akce, které mají být vykonány s každým hodinovým cyklem (zpracování instrukce, přerušení atd.)

Instrukce se popisují formou elementů a jsou sdružovány do skupin. Element je základní stavební blok jazyka CodAL a obsahuje následující sekce:

- Assembler - Tato sekce specifikuje textovou reprezentaci elementu ve formě řetězce, který je používán ke generování předpisu v assembleru.
- Binary - Sekce binary definuje reprezentaci instrukce nebo její části ve formě binárního kódování.
- Semantics - V této sekci je popsáno chování elementů či událostí v podmnožině ANSI C. Zakázány jsou následující prvky:
  - ukazatele
  - struktury a výčtové typy
  - příkaz goto
  - deklarace a inicializace proměnné v jednom příkaze (int i = 0;)
  - příkaz switch s jinými sekcemi než case a default
- Return - Tato sekce specifikuje výraz, který reprezentuje návratovou hodnotu elementu v ANSI C.
- Timing - Sekce timing je důležitá, když chceme popsat model procesoru na úrovni cyklů. Obsahuje seznam událostí, které mají proběhnout při aktivaci události.
- Start - Představuje hlavní prvek obsahující všechny instrukce implementované v modelu procesoru. Při zpracování instrukce se zde začínají hledat skupiny a v nich elementy jí odpovídající. Tato sekce je nejčastěji užívána pro vytváření assembleru a disassembleru.
- Decoders - Popisuje architekturu instrukčního dekodéru, má přímý vliv na vytvářený hardware a musí obsahovat následující informace:
  - adresa instrukce
  - zdroj, ve kterém jsou instrukce uloženy
  - jak zdroj adresovat
  - které dekodéry mají instrukci zpracovat

Kapitola 6

Codasip Framework

Informace v této kapitole vycházejí z [7].

Codasip Framework zpracovává jazyk CodAL a poskytuje integrované vývojové prostředí pro návrh procesorů s aplikačně specifickou instrukční sadou a víceprocesorových systémů na čipu (MPSoC). Jeho výhodou je automatizování procesů spojených s návrhem.

Poskytuje dva typy generátorů. První z nich slouží ke generování toolchainu, který by jinak musel být tvořen programátorem. Tím dramaticky zkracuje čas vývoje procesoru a šetří výlohy na něj vynaložené. Druhý generátor je hardwarový a jeho výstupem je plně syntetizovatelný RTL návrh procesoru.

Toolchain je využíván v průběhu návrhu procesoru, když je potřeba otestovat úpravu zdrojového kódu. Codasip toolchainem rozumíme sadu nástrojů využívaných v procesu vývoje modelované architektury:

- Assembler
- Překladač jazyka C
- Simulátor
- Profiler
- Debugger
- Disassembler

Assembler je nástroj, který překládá zdrojový kód assembleru do binárního objektového souboru. Codasip Assembler naráz pracuje se dvěma jazyky. Jedním je jazyk popsaný syntaxí instrukcí modelovaného procesu, druhým pak jazyk, který specifikuje direktivy a symboly.

Linker dohromady pojí binární objektové knihovny a doplnuje adresy, které v době překladu nebyly známy. Tento nástroj je nezávislý na architektuře procesoru, proto není součástí generovaného toolchainu.

Překladač jazyka C je ve fázi experimentálního návrhu a není plně podporován. Je tvořen z popisu instrukční sady ve dvou krocích. Nejdříve je z popisovaného modelu procesoru

extrahována a analyzována sekce semantics. Výsledkem je sémantika jednotlivých instrukcí, ze které je po možných úpravách návrhářem automatizovaně vytvořen compiler.

Simulátor dovoluje odhalit chyby v návrhu procesoru kontrolou zdrojů. Codasip Framework vytváří několik různých typů simulátoru lišících se složitostí.

Prvním z generovaných simulátorů je simulátor interpretovaný. Je založen na opakovaném načtení, dekódování a zpracování instrukcí z paměti. Umí pracovat ve dvou úrovních přesnosti, a to instrukcí a cyklů. V případě simulace s instrukční přesností, kdy je základním krokem instrukce, pracuje simulátor velice svižně, ale zanedbává spojitost s mikroarchitekturou procesoru, která není nijak zohledněná. Oproti tomu simulace s přesností cyklů krokuje pomocí taktu hodin, zohledňuje hardware, a proto je pomalejší.

Kompilovaný simulátor také pracuje na úrovni instrukcí a cyklů, je oproti interpretované verzi rychlejší, ale jeho vytvoření trvá déle. Je generován ve dvou krocích. V prvním se zanalyzuje simulovaná aplikace a je vytvořen jí odpovídající předpis v jazyce ANSI C. Ten je následně spolu se statickými zdroji procesoru kompilován do formy simulátoru.

Překládaný simulátor je z generovaných simulátorů nejvíce optimalizovaný a nejrychlejší. Vylepšuje kompilovaný simulátor, ale potřebuje dodatečné informace. Těmi jsou adresy začátků a konců všech základních bloků simulované aplikace.

RTL simulátor, který je posledním ze čtveřice, dokáže přesně simulovat hardware generovaný z popsaného modelu, pracuje však ze všech nejpomaleji.

Když jsou potřeba ještě podrobnější údaje, přichází na řadu profiler. Jedná se o klíčový nástroj při návrhu aplikačně specifických procesorů, jenž zpracovává a zaznamenává důležité jevy uvnitř modelu, které se odehrály po čas běhu programu. Mezi získané informace se řadí seznam nejčastěji používaných instrukcí, doba výpočtu, průchod kódem a další. Tím se vývojáři otevírá možnost cílové optimalizace konkrétních instrukcí a částí kódu, které se často provádějí.

Běží na úrovni instrukcí nebo cyklů a poskytuje následující funkčnost:

- nastavení breakpointů - jednoduché, podmíněné a datové
- změna hodnot zdrojů
- čtení hodnot zdrojů
- vyhodnocení výrazů

Disassembler je komplementární nástroj k assembleru. Zpracovává binární objektový soubor a tvoří z něj zdrojový kód assembleru.

Nyní, když jsme popsali jazyk a vývojové prostředí, se zaměříme na modelovaný procesor.

Kapitola 7

Popis architektury PowerPC

Informace v této kapitole vycházejí z [3, 4, 1].

PowerPC je rodina procesorů, která byla vytvořena roku 1991 koncorsiem firem Apple, IBM a Motorola. Největší slávu PowerPC zažilo jako procesor počítačů Macintosh firmy Apple. V současné době je využíván například v otevřené platformě Pegasos.

Tato superskalární RISC architektura našla díky své flexibilitě a škálovatelnosti místo v širokém spektru systémů, jako jsou osobní počítače, vestavěná zařízení, ale i vysoce výkonné sálové počítače pro vědecké výpočty a zpracování grafiky.

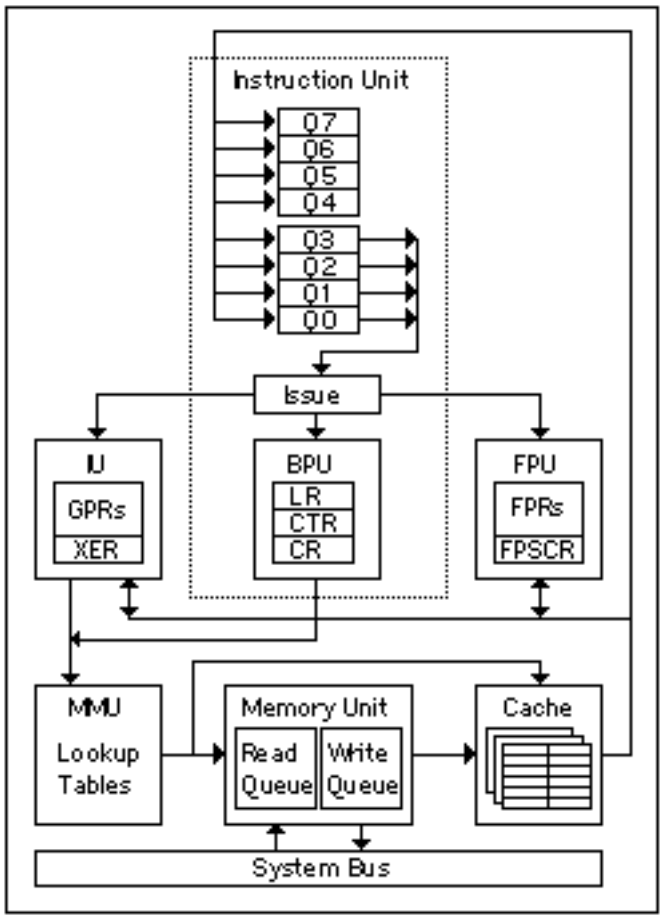
Implicitně pracuje s big-endian řazením bajtů, ale podporuje i režim little-endian. Díky tomu může být součástí různých víceprocesorových systémů či periferních zařízení navržených pro jiné procesory.

V následujících kapitolách budou popsány jednotlivé architektonické prvky architektury.

7.1 Funkční jednotky

Superskalární architektura PowerPC dosahuje vyššího počtu zpracovaných instrukcí za jednotku času tím, že obsahuje paralelní autonomní funkční jednotky řízené vlastním procesorem s vlastní instrukční sadou.

- Instruction Unit - jednotka zpracování instrukcí
- Branch Unit (BU) - jednotka zpracování skoků
- Integer Unit (IU) - jednotka pro operace s celými čísly
- Floating-point Unit (FPU) - jednotka pro operace v plovoucí řádové čárce
- Memory Management Unit (MMU) - jednotka organizující paměť
- Memory Unit (MU) - jednotka spravující přenos dat mezi pamětí a cache



Obrázek 7.1: Schéma zapojení funkčních jednotek PowerPC

7.1.1 Instruction Unit

Instrukce čekající na zpracování jsou řazeny do osmiprvkové fronty. Pokud se nacházejí v paměti cache, může být celá fronta naplněna naráz během jednoho taktu hodin procesoru.

7.1.2 Branch Unit (BU)

Tato část architektury PowerPC se stará o zpracování všech skokových instrukcí.

Všechny instrukce načtené z cache směřují skrze frontu do BU. Ta s předstihem kontroluje výskyt instrukcí skoků, a pokud je nalezne, odstraní je a na jejich místo vloží instrukce nalézající se na odpovídající adrese. Tento mechanismus se nazývá skládání skoků (branch folding) a má za následek, že jsou fronty IU a FPU vždy naplněné. Tím se účinně bojuje proti ztrátám výkonu během skoků, které jsou typickým problémem. V ideálním případě se totiž nad skoky nestráví vůbec žádný čas.

To, zda bude podmíněný skok pravděpodobně vykonán, se určuje kontrolou bitu CR, který je nastaven alespoň tři instrukce před skokem. Díky včasnému zamítnutí má BU dostatek času na načtení předpokládaných instrukcí.

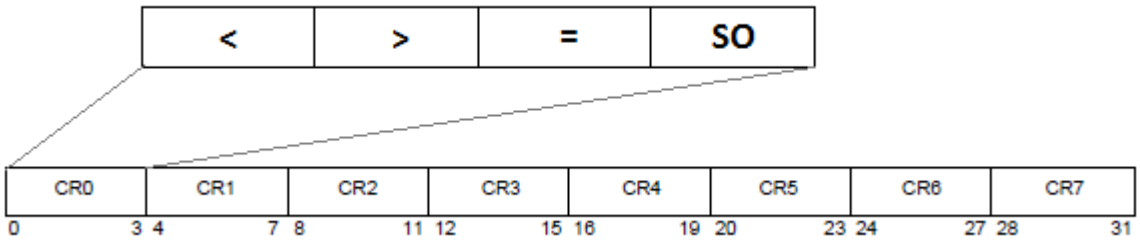
Druhým urychlujícím mechanismem používaným BU je provádění instrukcí mimo pořadí (out of order execution). Když je ve frontě detekována instrukce s číslem v plovoucí řádové čárce a FPU je nečinná, je z fronty vyjmuta a poslána ke zpracování. Stejný princip platí i pro instrukce, které ovlivňují BU. Je zaručeno, že tato heuristika nebude použita, pokud by instrukce závisela na jiné ve frontě před ní.

Condition Register (CR)

Na tento 32-bitový registr můžeme považovat za pole osmi prvků po čtyřech bitech nebo za jednotlivé bity.

Jeho hodnota je aktualizována, když má instrukce (pokud tuto volbu podporuje) nastavena record bit. Ten, který se v assemblerovské syntaxi zadává přidáním tečky za název instrukce. Implicitně je pro aktualizaci vybráno CR0 pro integer instrukce a CR1 pro floating-point instrukce. Přesto se dá zvolit libovolné z osmi polí zadáním jeho indexu jako prvního parametru instrukce.

První tři bity pole jsou nastaveny podle porovnání hodnot operandů, čtvrtý je kopírován z XER a určuje, zda proběhlo přetečení.



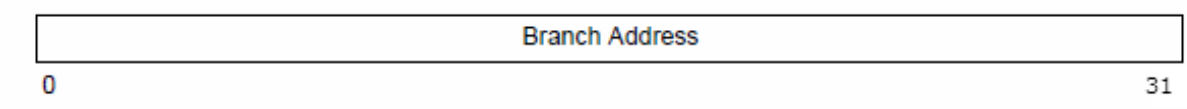
Obrázek 7.2: Condition Register

Výhodou většího počtu na sobě nezávislých polí je, že může proběhnout více početních operací, aniž by bylo potřeba čekat na kontrolu výsledku.

Link Register (LR)

Na rozdíl od ostatních architektur neukládá PowerPC návratovou adresu na zásobník, ale má pro ní zřízen speciální registr.

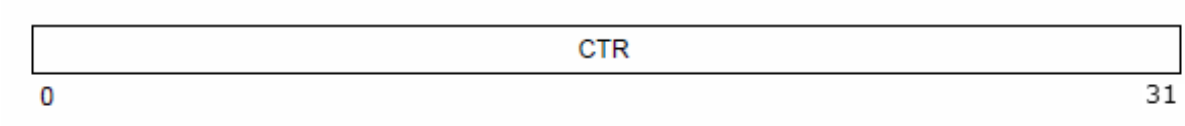
Všechny instrukce skoků mají možnost mít nastaven link bit, který se v assemblerovském kódu zadá přidáním písmene l. Pokud je tento bit nastaven, LR obdrží adresu instrukce následující po skoku.



Obrázek 7.3: Link Register

Count Register (CTR)

Tento víceúčelový registr může být použit pro počítání průchodů smyčkou (cyklus for) nebo pro uchování adresy a následný skok (instrukce bctr).



Obrázek 7.4: Count Register

7.1.3 Integer Unit (UI)

Tato jednotka zpracovává všechny aritmetické a logické operace, posuvy, rotace, porovnání i čtení a zápis celých čísel do paměti.

Podporuje techniku feed-forwarding, při které se výsledek ALU operace okamžitě poskytuje instrukci, která na něj čeká (souvisle se zápisem do GPR). Také vypočítává adresy pro všechny ostatní funkční bloky.

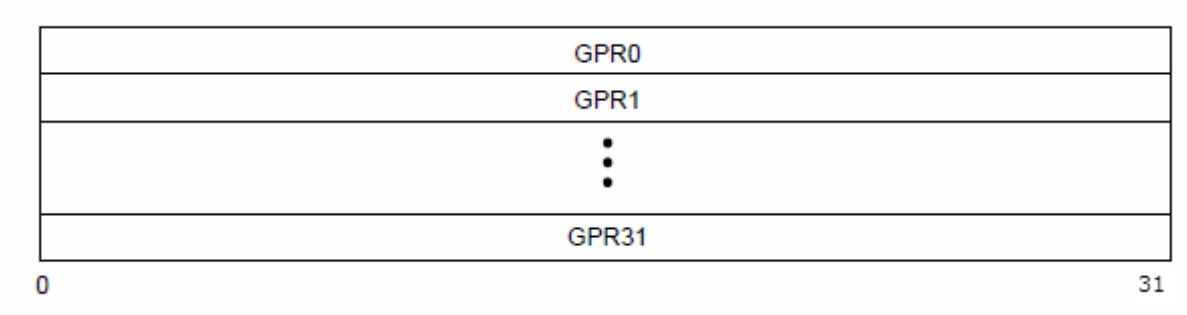
32x General Purpose Register

Sada registrů obecného užití sloužící k uchovávání hodnot proměnných, parametrů, ukazatelů a dalších celých čísel.

IU pracuje se speciálními instrukcemi pro datové přesuny mezi GPR, Link a Count registru nacházejících se pod správou Branch Unit.

Speciální chování vykazuje GPR0, který místo svého obsahu zastupuje v některých instrukcích hodnotu 0.

- GPR0 - ukládá se do něj LR při budování rámce zásobníku
- GPR1 - ukazatel na zásobník
- GPR2 - ukazatel na Table Of Contents (globální proměnné, ukazatele na funkce)
- GPR3 - návratová hodnota, první argument funkce
- GPR4-10 - druhý až osmý argument funkce
- GPR11-31 - bez zvláštního významu



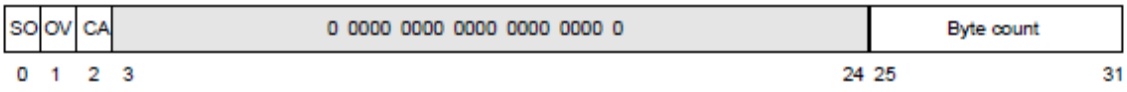
Obrázek 7.5: General Purpose Register

Aritmetic Logic Unit (ALU)

Aritmeticko-logická jednotka získává instrukce skrze frontu Branch Unit. Implementuje obvyklé aritmetické a logické operace, posuvy, rotace a porovnání čísel.

Exception Register (XER)

XER je registr speciálního významu. Udržuje informace o přetečení a přesunu během operací v ALU a obsahuje pole, do kterého se zadává délka řetězce pro jeho čtení z paměti nebo zápis do paměti.



Obrázek 7.6: Exception Register

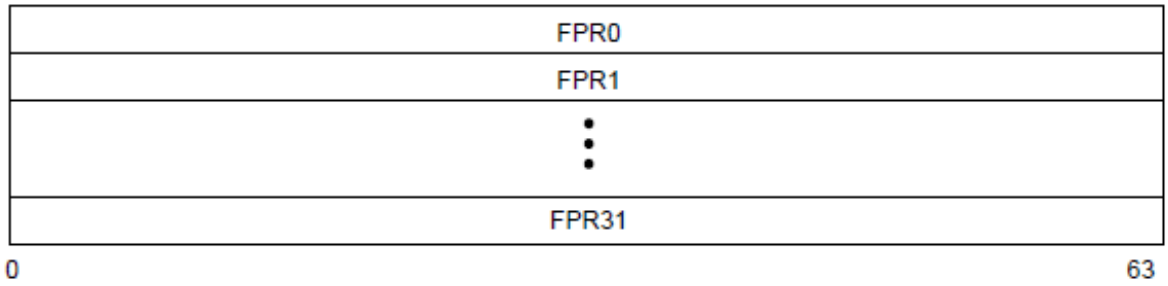
7.1.4 Floating-Point Unit (FPU)

Jednotka pro práci s čísly v plovoucí řádové čárce provádí veškeré artimetické operace, porovnání, konverze a manipulace. Na rozdíl od IU nepodporuje feed-forwarding.

32x Floating-Point Register

Sada 64 bitových registrů s podporou čísel s jednoduchou a dvojitou přesností.

- FPR0 - bez zvláštního významu
- FPR1 - návratová hodnota, první argument funkce
- FPR2-13 - druhý až třináctý argument funkce
- FPR14-31 - bez zvláštního významu



Obrázek 7.7: Floating-Point Register

Aritmetické jednotky MAC a DIV

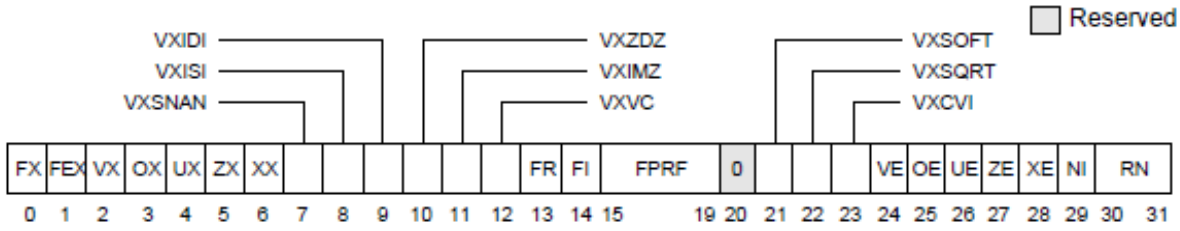
Architektura PowerPC pochází ze sálových počítačů, jejichž typickými operacemi bylo násobení a sčítání, proto má dvě oddělené aritmetické jednotky Multiply-and-Accumulate Unit (MAC) a děličku (DIV).

MAC je uzpůsobená k násobení a přičtení hodnot v jednom kroku. Tyto operace jsou využívány při práci s vektory a maticemi během grafických transformací a pro aproximaci hodnoty rozvojem. Pro samotné sčítání dvou čísel se násobí první jedničkou a pro samotné násobení se k výsledku přičítá nula.

Dělička pracuje pomaleji, proto optimalizátory často mění dělení na násobení předem připravenou převrácenou hodnotou.

Floating-Point Status Control Register (FPSCR)

Jsou do něj ukládány informace o výsledku a způsobených výjimkách podle standardu IEEE.



Obrázek 7.8: Floating-Point Status and Control Register

7.1.5 Memory Management Unit (MMU)

Stará se o překlad z logické na fyzickou adresu, přístupová práva, ochranu paměti a stránkování virtuální paměti. Výkonnost je uchováváním posledně použitých adres v tabulkách.

7.1.6 Memory Unit (MU)

Zabezpečuje přenos dat mezi cache a pamětí. Obsahuje dvě fronty pro čtení a tři pro zápis, každá má kapacitu sektoru cache (8 slov). Zápis do paměti je nejčastěji prováděn, aby se v cache udělalo místo pro nová data. Nejpozději použitý (LRU) sektor je přesunut do fronty pro zápis, ze které je poslán na systémovou sběrnici.

7.2 Paměť

Počítače PowerPC mají Von Neumannovskou architekturu, jejíž typickým rysem je společná paměť pro instrukce a data. Velikost paměti závisí na konkrétním modelu.

7.3 Instrukční sada

Instrukční sada procesorů PowerPC byla navržena tak, aby jednotlivé instrukce byly co nejjednodušší a měly velikost slova. Díky uchování všech potřebných informací na jednom místě je jejich dekódování a provedení velice rychlé - každá z částí trvá pouhý jeden takt vnitřních hodin. Nutnost zakódovat všechny operandy do binární reprezentace instrukce klade omezení na výsledný počet GPR.

Další z výhod je použití tříadresného formátu aritmetických instrukcí, který zmenšuje režii při kopírování obsahů registrů, neboť hodnoty použité jako zdroje nejsou přepisovány.

PowerPC patří do skupiny LOAD/STORE architektur. Ty se vyznačují tím, že jedinými manipulacemi s pamětí může být ukládání a čtení. Ostatní operace pracují s daty uloženými v rychlých registrech. Výjimkou jsou takzvané bezprostřední (immediate) operandy, které jsou zakódovány v binárním zápisu instrukce.

7.3.1 Instrukční sady funkčních jednotek

Tato podkapitola stručně shrnuje instrukční sadu. Architektura PowerPC obsahuje tři kategorie uživatelských instrukcí, které jsou zpracovávány příslušnou funkční jednotkou.

Integer Unit

- Aritmetika - sčítání, odčítání, násobení, dělení
- Logika - AND, OR, XOR, NAND, NOR, NOT, EQV, rozšíření znaménka, počet vedoucích nul
- Posuvy/Rotace - vyjmutí, vložení, smazání, posuvy, rotace
- Čtení/Zápis - z/do LR, CR, CTR, GPR či FPR

Floating-Point Unit

- Aritmetika - sčítání, odčítání, násobení, dělení, násobení-sčítání, násobení-odčítání, negace, absolutní hodnota
- Konverze - zaokrouhlení, převod na celá čísla
- Porovnávání

- Čtení/Zápis - z/do FPSCR či FPR

Branch Unit

- Skoky - podmíněné a nepodmíněné
- Systémová volání
- Manipulace - CR

7.3.2 Rozpoznání instrukce

V prvních šesti bitech instrukce je uložen její operační kód (opcode). Ten buď přímo identifikuje instrukci, nebo skupinu instrukcí. V druhém případě je potřeba určit, která instrukce bude ze skupiny vybrána. K tomu slouží rozšířený operační kód (extended opcode), který je zakódován v binární reprezentaci instrukcí na pozicích udaných předpisem tvaru konkrétního prvku.

Tento způsob identifikace je společný napříč celou instrukční sadou, ostatní vlastnosti se pevně váží ke konkrétním skupinám instrukcí.

Z následujícího obrázku je patrné, že pokud je instrukce jednoznačně identifikována pomocí operačního kódu, není rozšířený operační kód potřeba. Důvod jeho existence je ten, že instrukční sada má větší velikost, než lze jednoznačně umístit do šesti bitů. Díky udržení obecné identifikace v tomto rozsahu mohou některé instrukce plně využívat zbylý binární obsah na zakódování potřebných parametrů.

	OPCODE				EXTENDED OPCODE		
add*x*	31	D	A	B	OE	266	Rc
addc*x*	31	D	A	B	OE	10	Rc
adde*x*	31	D	A	B	OE	138	Rc
addi	14	D	A	SIMM			
addic	12	D	A	SIMM			

Obrázek 7.9: Demonstrace operačního a rozšířeného operačního kódu

7.3.3 Adresní módy instrukcí

Instrukce skoků v sobě musí mít zakódovány jednoznačný identifikátor a cílovou adresu. Procesory PowerPC pracují s adresováním paměti relativně k programovému čítači a omezeně podporují absolutní adresování.

Je požadováno, aby instrukce v paměti byly zarovnány podle slov. Za tohoto předpokladu je možné dva nejméně významné bity adresy skoku považovat za nulové a adresovat po instrukcích (čtveřicích bajtů). Jak velký rozsah můžeme adresovat?

- Podmíněný skok - 14 bitů pro adresování, podporuje rozsah +-32 KB.
- Nepodmíněný skok - 24 bitů pro adresování, podporuje rozsah +- 32MB.

Daná adresa je považována za absolutní, pokud má instrukce nastaven příznak AA (absolute adress). Poté je možné adresovat horních a dolních 32 KB paměti v případě podmíněného a horních či dolních 32 MB paměti v případě nepodmíněného skoku. Přesto se toto absolutní adresování téměř nevyužívá.

Existuje totiž mechanismus, díky němuž lze adresovat jakékoli místo v paměti pomocí ukazatele. Ten je přenesen do Link nebo Count registru a speciální instrukcí je proveden skok na danou adresu. Získání tohoto ukazatele je však náročnější a provádí se ve více krocích. Horní a dolní polovina ukazatele je zakódována jako šestnáctibitová bezprostřední hodnota aritmetických instrukcí, které je nutno spojit za pomocí logických posuvů a součtů.

7.3.4 Adresní módy dat

PowerPC umožňuje adresovat data dvěma způsoby: D-Form adresování a X-Form adresování.

D-Form adresování (displacement-based adressing) vypočítává adresu paměti ze součtu obsahu GPR a šestnáctibitové znaménkové bezprostřední hodnoty. Na rozdíl od adres instrukcí nejsou na adresy dat kladeny nároky na zarovnání, proto výsledná adresa udává pozici v bajtech.

Speciální vlastnost má GPR0, jehož obsah je vždy nulový.

```
r0 = 500
r1 = 1000

stw r2, 16(r0) // Uloží obsah registru r2 na adresu 16.
stw r2, 16(r1) // Uloží obsah registru r2 na adresu 1016.
```

X-Form adresování (index-based adressing) využívá jeden GPR jako bázi a druhý jako index. Jejich obsahy tvoří výslednou adresu, která dovoluje adresování většího kusu paměti než D-Form. Stejně jako u něj je bázový registr GPR0 nahrazen nulou. Pro registr sloužící jako index tato vlastnost neplatí.

```
r0 = 500
r1 = 1000

stwx r2, r1, r0 // Uloží obsah registru r2 na adresu r1+r0.
stwx r2, r0, r1 // Uloží obsah registru r2 na adresu 0+r1.
```

V této kapitole byly popsány klíčové prvky a vlastnosti procesoru PowerPC, nyní tyto znalosti namodelujeme.

Kapitola 8

Implementace

Procesor PowerPC byl modelován na instrukční úrovni. Podrobnější vlastnosti, jako je například vnitřní časování, jsou opomenuty. Díky tomu je model jednodušší, lze ho snadněji popsat a automatizované vytváření nástrojů je svižnější. Generovaný simulátor pracuje s nejmenším krokem jedné instrukce a je velice rychlý.

Protože tento typ modelu zanedbává přesný popis vnitřní architektury, ztrácí spojitost s reálným procesorem a export do VHDL není jednoduchý.

Při konzultacích s vedoucím práce bylo dohodnuto, že modelování Floating-Point Unit není nezbytné, proto tato jednotka nebyla modelována.

8.1 Popis zdrojů

Přesto, že model na instrukční úrovni zanedbává určité aspekty architektury, zdroje musí být popsány, neboť jsou využívány při výpočtech, a tudíž úzce svázány s modelem.

8.1.1 Registry

PowerPC na úrovni uživatelské instrukční sady obsahuje 32 registrů obecného účelu (GPR), které je příhodné modelovat jako pole, a sadu registrů se speciálním významem (LR, CR, CTR, XER). Tyto registry mají předponu "arch", která označuje příslušnost k modelované architektuře.

Pro vlastní potřeby zpracování instrukcí jsou modelovány další dva registry. První z nich slouží jako buffer pro uchování instrukce, druhý pak jako záloha programového čítače, který je díky pořadí prováděných operací přepsán před provedením instrukce. Můžeme na ně pohlížet spíše jako na globální proměnné.

```
arch register bit[32] gpregs[32];         // General Purpose Registers
arch register bit[32] cr;                 // Condition Register
arch register bit[32] lr;                 // Link Register
arch register bit[32] ctr;                // Count Register
arch register bit[32] xer;                // Fixed Point Exception Register

register bit[32] fetched_instr;           // Instruction Buffer
register bit[32] fetched_pc;              // Program Counter Back-up
```

Jazyk CodAL vyžaduje explicitní modelování programového čítače jako zvláštního registru.

```
program_counter bit[32] pc;                // Program Counter
```

8.1.2 Paměť

Pro implementaci je použita speciální konstrukce "memory". Kombinace jejích parametrů určuje výsledné vlastnosti paměti.

Paměť procesorů PowerPC je typu Big-Endian, má šířku slova (32 bitů) a je adresovatelná po bajtech. Jedinými operacemi, které lze s pamětí provádět, je čtení a zápis, každá z nich trvá jeden instrukční takt.

```
memory bit[32] mem {
    .latency = {1, 1},
    .endianess = big,
    .lau = 8,
    .size = 0x200000,
    .flags = {r, w}
};
```

Na závěr musíme namapovat paměť na sběrnici.

```
memorymapping defaultmap {
    0x0.. 0x007FFFFF = mem[31..0];
};
```

8.2 Operandy

Po definování hardwarových zdrojů pokračujeme modelováním instrukčních operandů, které jsou popisovány formou elementů. Ty lze považovat za základní stavební jednotky různého určení. Elementem je například operand, instrukce nebo pouze její operační kód.

V následujícím příkladu bude demonstrováno definování modifikátorů instrukcí skoků, které určují, zda bude adresa operandu považována za absolutní či relativní.

```
element modifier_aa_off {
    assembler { "" };
    binary { 0b0 };
    return { 0; };
}

element modifier_aa_on {
    assembler { "a" };
    binary { 0b1 };
    return { 1; };
}
```

O stupeň abstraktnějším prvkem je množina (set), která reprezentuje všechny elementy v ní obsažené. Vzhledem k nutnosti jednoznačnosti modelu nesmí množina obsahovat dva totožně se chovající elementy.

```
set modifier_aa = modifier_aa_off, modifier_aa_on;
```

Množina může sestávat i z jiných množin. Tento mechanismus dovoluje hierarchické řazení sémanticky podobných instrukcí do skupin, ze kterých je ve finále vytvořena množina reprezentující úplnou instrukční sadu.

8.3 Zpracování instrukcí

Pro vytvoření funkčního modelu je potřeba modelovat události spojené se zpracováním instrukcí. V této kapitole jsou uvedeny příklady klíčových událostí (Reset, Main a Halt) potřebných pro zprovoznění modelu PowerPC. Tento proces je díky předpřipraveným konstrukcím jazyka CodAL velice efektivní a rychlý.

Událost Reset definuje akce, které mají být nezbytně vykonány, pokud je procesor v počátečním stavu simulace. Jak je z následujícího kódu patrné, jedná se o inicializace registrů a nastavení programového čítače na začátek programu.

```
#define REGISTER_COUNT 32
event reset {
    semantics {
        int i;
        pc = 0;
        cr = 0;
        lr = 0;
        ctr = 0;
        xer = 0;

        for(i = 0; i < REGISTER_COUNT; i++) {
            gpregs[i] = 0;
        }
    };
}
```

Událost Main je vykonávána s každým hodinovým cyklem. Díky tomu, že popisujeme RISC procesor, je ideálním adeptem pro zpracování a provádění instrukcí spolu s inkrementací programového čítače.

Sekce Start určuje počáteční místo pro dekódování instrukcí assemblerem a překladačem.

Sekce Decoders je používána simulátorem a generátorem popisu hardware. Argument této sekce určuje registr, ve kterém je uložena adresa instrukce určená pro dekódování. Vnitřní instance reprezentuje kompletní instrukční sadu, jejímž parametrem je registr obsahující načtenou instrukci.

```
#define INSTRUCTION_BYTE_SIZE 4
event main {
    use instructions_powerpc_all as instr;

    start {{ instr; }};
    decoders (fetched_pc) {{ instr(fetched_instr); }};
```

```
    semantics {
        fetched_instr = mem[pc];
        fetched_pc = pc;
        pc = pc + INSTRUCTION_BYTE_SIZE;
    };
}
```

V poslední řadě je třeba popsat událost, kterou simulace skončí. V takto určené události můžeme vytisknout obsahy registrů, paměti, hodnotu programového čítače a jiné.

V implementaci této události halt pro PowerPC je použita funkce, která ukládá návratovou hodnotu simulovaného programu do souboru "sim_exit_code"v aktuálním adresáři. Tento mechanismus je potřebný pro automatizované testování, neboť se návratový kód programu a simulátoru liší.

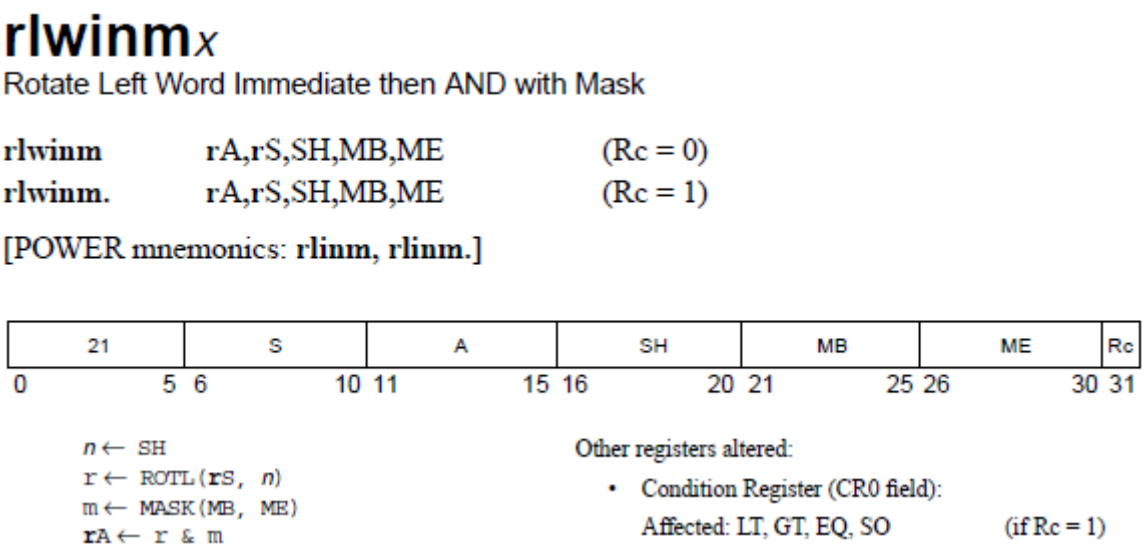
```
event halt {
    semantics {
        codasip_store_exit_code((int)gpregs[3]);
    };
}
```

8.4 Modelování instrukce

Nyní, když máme popsané zdroje architektury, elementy a způsob zpracování instrukcí, můžeme přistoupit k jejich modelování.

Popis procesoru PowerPC na úrovni instrukční sady nedovoluje detailně modelovat vzorový hardware. Proto je rozdělení instrukcí podle původních funkčních jednotek spíše ilustrativní. Ve skutečnosti jsou instrukční sady všech nezávislých jednotek procesoru spojeny v jeden celek.

Pro účely demonstrace modelování byla vybrána instrukce RLWINM (Rotate Left Word Immediate then AND with Mask) patřící do instrukční sady Integer Unit.



rlwinmx

Rotate Left Word Immediate then AND with Mask

rlwinm	rA,rS,SH,MB,ME	(Rc = 0)
rlwinm.	rA,rS,SH,MB,ME	(Rc = 1)

[POWER mnemonics: **rlinm, rlinm.**]

21	S	A	SH	MB	ME	Rc
0 5	6 10	11 15	16 20	21 25	26 30	31

```
n ← SH
r ← ROTL(rS, n)
m ← MASK(MB, ME)
rA ← r & m
```

Other registers altered:

- Condition Register (CR0 field):
  Affected: LT, GT, EQ, SO (if Rc = 1)

Obrázek 8.1: Přehled vlastností instrukce RLWINM

Při analýze instrukce zjistíme, že pracuje se dvěma obecnými registry, z nichž jeden má úlohu zdroje (rS) a druhý je použit pro uložení výsledku (rA). Funkce jako třetí až pátý parametr (SH, MB, ME) přijímá celá čísla bez znaménka o velikosti 5 bitů, tedy v rozsahu 0 až 31. Existuje však ještě jeden modifikátor, který je reprezentován tečkou za názvem instrukce. Ten je v binárním předpisu umístěn na pozici nejméně významného bitu a určuje, zda se o výsledku mají uložit souhrnné informace do prvního pole CR.

Nejdříve namodelujeme název a operační kód instrukce.

```
#define OP_RLWINM      0x15 // 21
element op_rlwinm {
    assembler { "rlwinm" };
    binary { OP_RLWINM:6 };    // Identifikátor bude zakódován na 6 bitů
    return { OP_RLWINM; };
}
```

Následně parametry instrukce (sadu registrů, operandy a Record bit).

```
// Sada registrů
element gpr represents gpregs {
    assembler { index=unsigned };
    binary { index=0b[5] };
    return { index; };
}

// Celé číslo bez znaménka na pěti bitech
element uimm5 {
    assembler { immval=unsigned };
    binary { immval=0b[5] };
    return { immval; };
}

// Record bit
element modifier_rc_off {
    assembler { "" };
    binary { 0b0 };
    return { 0; };
}
element modifier_rc_on {
    assembler { "." };
    binary { 0b1 };
    return { 1; };
}
set modifier_rc = modifier_rc_off, modifier_rc_on;
```

Nyní, když máme připraveno vše potřebné pro implementaci kýžené instrukce, můžeme ji začít modelovat. (Tento zápis je převzat ze zdrojového kódu modelu PowerPC a zjednodušen pro potřeby demonstrace.)

```
// Definice elementu instrukce
element instruction_rlwinm {
    // Použité podčásti instrukce
    use op_rlwinm;
    use modifier_rc;
    use gpr as reg_s, reg_a;
    use uimm5 as sh, mb, me;

    // Zápis instrukce v assembleru ("~" značí konkatenaci).
    assembler { op_rlwinm ~ modifier_rc reg_a "," reg_s "," sh "," mb "," me };
    // Kódování instrukce do binární reprezenatace.
    binary { op_rlwinm reg_s reg_a sh mb me modifier_rc };

    // Jazykem ANSI C je popsána sémantika odpovídající obrázku 6.1
    semantics {
        int result, mask;

        // Provedení rotace makrem
        result = ROTATE_LEFT(gpregs[reg_s], sh);
        // Nastaveni masky
        mask = MASK(mb, me);
        // Zápis výsledné hodnoty do cílového registru
        RWRITE(reg_a, result & mask);
        if(modifier_rc) {
            SET_CR_FIELD(CR0, result & mask);
        }
}
```

8.5 Modelování aliasu

Alias neboli mnemotechnická pomůcka slouží programátorům ke snadnějšímu zorientování v assemblerovském programu při zachování stejného binárního kódování.

Instrukce ADD má dva parametry: registr a hodnotu, která k němu má být přičtena. Velice častou operací je inkrementování proměnné o jedničku, proto byl vytvořen alias INC, který je zakódován jako ADD s druhým parametrem rovným jedné.

```
inc reg7 <=> add reg7, 1
```

Funkce RLWINM byla pro demonstraci modelování vybrána mimo jiné z důvodu, že má největší počet mnemotechnických pomůcek.

Jedním z nich je alias ROTLWI, který lze modelovat následovně:

```
rotlwi rA,rS,n <=> rlwinm rA,rS,n,0,31

element op_rotlwi_alias {
    assembler { "rotlwi" };
```

```
    binary { OP_RLWINM:6 };
    return { OP_RLWINM; };
}

element instr_rotlwi_alias {
    use gpr as reg_s, reg_a;
    use op_rotlwi_alias as op_rotlwi;
    use uimm5 as n;
    use modifier_rc;

    assembler { op_rotlwi ~ modifier_rc reg_a "," reg_s "," n };
    binary { op_rotlwi reg_s reg_a n 0:5 31:5 modifier_rc};
}
```

Existují však i aliasy, které za stávající situace modelovat nelze. Důvodem je, že vyžadují aritmetické operace v sekci binary, které v současné době nejsou podporované. Po dohodě s vedoucím práce byly jejich implementace ponechány zakomentované, neboť je tato funkcionalita brzy plánována. Do té doby jsou všechny assemblerovské programy pro PowerPC upravovány automatizovaným skriptem v jazyce Python a za pomoci regulárních výrazů jsou tyto aliasy identifikovány a překonvertovány do podporované formy.

Příkladem aliasu z této skupiny je instrukce SLWI:

```
slwi rA,rS,n <=> rlwinm rA,rS,n,0,31-n
```

Kapitola 9

Simulace

Správnou funkčnost svého modelu jsem ověřil sadou testů v jazyce C, které mi pro tento účel byly poskytnuty výzkumnou skupinou Lissom.

Za pomoci tohoto balíčku stovek specializovaných testů různých úrovní obtížností jsem laděním assemblerovského kódu verifikoval 71 z 107 instrukcí. Zbylou část instrukční sady jsem otestoval formou kontroly jednotlivých instrukcí v několika rozdílných případech, lišících se generováním příznaků v CR a jiných registrech.

Díky těmto pečlivým kontrolám jsem přesvědčen, že celý model pracuje správně a věrně emuluje procesor PowerPC na úrovni instrukční sady.

9.1 Výběr konkurenta

Kvalitu výsledného modelu určují mimo správného chování i další parametry. Z nich nás nejvíce zajímá rychlost, se kterou simulátor zpracuje a vykoná program, poskytující půdu pro srovnání efektivity a optimalizací výpočtů.

Vygenerovaný simulátor modelu jsem se rozhodl poměřit s výkonností PSIM, který je v současné době jedním z nejznámějších a nejkvalitnějších simulátorů instrukční sady procesorů PowerPC.

Díky svým vynikajícím vlastnostem se stal součástí GNU PowerPC Debuggeru, a tudíž se stále vyvíjí. Tento fakt je důležitý, neboť pro objektivní srovnání výkonnosti je účelné konfrontovat vygenerovaný simulátor se současnou konkurencí.

9.2 Testy

Po dohodě jsem vybral z široké sady tři reprezentativní testy pro účely porovnání výkonností obou simulátorů. Z důvodu eliminace náhodných poklesů výkonu procesoru při simulaci a souvisejícího ovlivnění výsledků jsem jednotlivé testy pouštěl s více iteracemi, aby výpočet trval déle a tyto výkyvy se rovnoměrně rozprostřely v čase. Pro každý z testů je prováděno pět měření a výsledek je zprůměrován.

- Bitcount - 10 000 iterací
- Dijkstrův algoritmus - 1 000 iterací
- Řadicí algoritmus Quicksort - 1 000 iterací

9.2.1 Bitcount

Program Bitcount počítá výskyt jedničkových bitů v poli 256 bezznaménkových integerů. Výsledek, upraven aritmetickým posunem doprava, musí odpovídat hodnotě 128.

9.2.2 Dijkstrův algoritmus

Dijkstrův algoritmus byl vymyšlen nizozemským informatikem Edsgerem Dijkstrou v roce 1959. Jedná se o nejrychlejší známý algoritmus pro nalezení všech nejkratších cest ze zadaného uzlu do ostatních uzlů grafu, který je kladně hranově ohodnocen.

Na Dijkstrův algoritmus lze pohlížet jako na zobecněné prohledávání grafu do šířky, při kterém se vlna nešíří na základě počtu hran od zdroje, ale vzdálenosti od zdroje (ve smyslu váhy hran). Tato vlna proto zpracovává jen ty uzly, k nimž již byla nalezena nejkratší cesta.

Dijkstrův algoritmus si uchovává všechny uzly v prioritní frontě řazené dle vzdálenosti od zdroje - v první iteraci má pouze zdroj vzdálenost 0, všechny ostatní uzly nekonečno. Algoritmus v každém svém kroku vybere z fronty uzel s nejvyšší prioritou (nejnižší vzdáleností od již zpracované části) a zařadí jej mezi zpracované uzly. Poté projde všechny jeho dosud nezpracované potomky, přidá je do fronty, nejsou-li tam již obsaženi, a ověří, zda-li nejsou blíže zdroji, než byli před zařazením právě vybraného uzlu mezi zpracované.

To znamená, že pro všechny potomky ověřuje:

```
vzdálenost_zpracovaný + délka_hrany(zpracovaný, potomek) < vzdálenost_potomek
```

Pokud nerovnost platí, tak danému potomkovi nastaví novou vzdálenost a označí za jeho předka zpracovávaný uzel. Po průchodu přes všechny potomky algoritmus vybere z fronty uzel s nejvyšší prioritou a celý krok opakuje. Algoritmus je ukončen v okamžiku, kdy jsou zpracovány všechny uzly (prioritní fronta je prázdná). [1]

Test pracuje se třiceti uzly a zjišťuje nejkratší vzdálenost (16 jednotek) mezi prvním a druhým z nich.

```
Dijkstra(0, 1) = 16. Posloupnost: 0->17->8->1
```

9.2.3 Řadící algoritmus Quicksort

Quicksort neboli rychlé (rekurzivní) řazení do tříd je jedním z nejrychlejších známých algoritmů řazení založených na porovnávání prvků. Jeho průměrná časová složitost je pro algoritmy této skupiny nejlepší možná $O(N * \log_2 N)$, v nejhorším případě je však jeho časová náročnost $O(N^2)$.

Základní myšlenkou quicksortu je rozdělení řazené posloupnosti čísel na dvě přibližně stejné části (quicksort patří mezi algoritmy typu rozděl a panuj). V jedné části jsou čísla větší a ve druhé menší, než nějaká zvolená hodnota (pivot). Pokud je zvolen dobře, jsou obě části přibližně stejně velké. Pokud budou obě části samostatně seřazeny, je seřazené i celé pole. Obě části se pak rekurzivně řadí stejným postupem. [2]

Tento test řadí pole 256 integerů, po seřazení vypočítá součet řady $\sum_{i=0}^{255} sorted\_tab[i] * i$. Nakonec provede operaci ekvivalence nad jednotlivými bajty výsledku. Návratová hodnota se musí rovnat 255.

[1] http://www.algoritmy.net/article/5108/Dijkstruv-algoritmus
[2] http://cs.wikipedia.org/wiki/Quicksort

9.3 Vyhodnocení simulace

Porovnávání probíhalo na počítači s těmito parametry:

- Procesor AMD Turion X2 2.2GHz, 2MB L2 Cache, FSB 400 MHz
- Paměť 4GB, 800 MHz
- Operační systém Windows 7, 64 bit

Jako vztažnou hodnotu pro porovnání výkonnosti používám mnou vygenerovaný simulátor bez profileru. Obecně platí, že čím je vyšší hodnota koeficientu výkonnosti, tím rychleji simulátor pracuje. Všechny testy byly zpracovány s nejvyšší úrovní optimalizace.

	PSIM	**Codasip PowerPC**		**Codasip PowerPC s profilerem**	
	[s]	[s]	[MHz]	[s]	[MHz]
Bitcount	16,62	40,28	13,35	638,40	0,84
Dijkstra	84,56	151,02	12,99	2288,06	0,86
Quicksort	27,32	46,88	13,05	747,81	0,82
průměr	42,83	79,38	13,13	1224,76	0,84
výkonnost	1,85	1		0,06	

Tabulka 9.1: Srovnání simulátorů

9.3.1 Porovnání simulátorů

Z výsledků měření vyplývá, že generovaný simulátor z modelu procesoru PowerPC je přibližně o polovinu pomalejší než konkurenční PSIM.

Jedním z možných důvodů je, že Codasip Framework vytváří univerzální simulátory, zatímco PSIM je vysoce optimalizován pro PowerPC.

Při srovnání frekvence našeho simulátoru s nejjednodušší PowerPC architekturou 601 (50-66MHz), docházíme k závěru, že výkonnost emulovaného procesoru je vůči reálnému přibližně čtvrtinová.

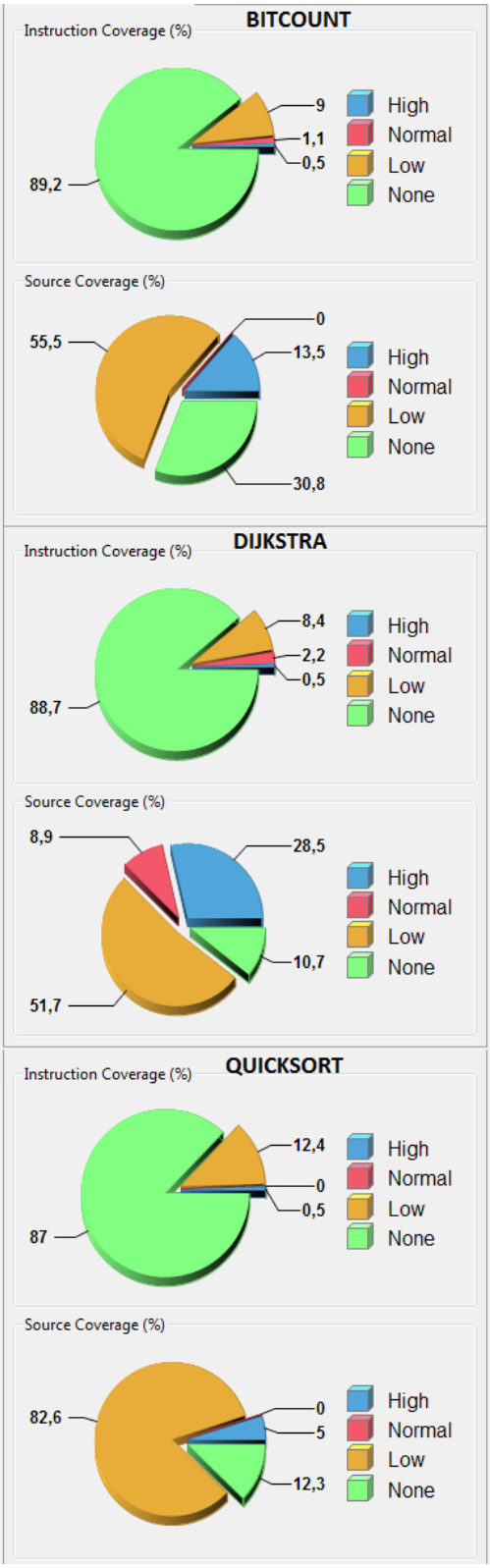
9.3.2 Simulace s profilerem

Aktivovaný profiler shromažďuje po čas simulovaní detailní informace o vnitřních procesech. Cenou za dodatečné znalosti je razantní zpomalení simulátoru na patnáctinu původní výkonnosti.

Získané informace odhalily zajímavosti. Klíčovou instrukcí je lwz (načtení slova z paměti do registru), která ve všech testech zaujala první pozici v počtu výskytů i době strávené výpočtem. Průměrně je každou třetí instrukcí programu, což koresponduje s load/store architekturami typu RISC. Při snaze o zlepšení výkonnosti simulátoru je spolu s komplementární instrukcí stw (uložení slova z registru do paměti) a instrukcí rlwinm (rotace s maskou) ideálním cílem, na který by bylo vhodné se zaměřit.

Pro mě nejzajímavějším faktem je, že každý z programů byl tvořen nejvýše třinácti opakujícími se instrukcemi, tedy pouze jednou sedminou celé instrukční sady.

Přestože byl assemblerovský kód generován pomocí optimalizovaného GNU překladače, program Bitcount obsahuje třetinu neprovedeného kódu. Ten byl očekáván pouze v případě větvení při podmínkových skocích.



Instruction	Count	Percent
Top by usage		
lwz gpr, <num> (<num>)	18435004	34.2
stw gpr, <num> (<num>)	9475005	17.6
addic gpr, <num> , <num>	8705000	16.1
cmpi <num> , <num> , gpr, <num>	4610001	8.5
bc <num> , <num> , <num>	4610001	8.5
Top by memory access		
lwz gpr, <num> (<num>)	36870008	44.9
stw gpr, <num> (<num>)	18950010	23.1
addic gpr, <num> , <num>	8705000	10.6
cmpi <num> , <num> , gpr, <num>	4610001	5.6
bc <num> , <num> , <num>	4610001	5.6
Top by delay		
lwz gpr, <num> (<num>)	18435004	34.2
stw gpr, <num> (<num>)	9475005	17.6
addic gpr, <num> , <num>	8705000	16.1
cmpi <num> , <num> , gpr, <num>	4610001	8.5
bc <num> , <num> , <num>	4610001	8.5

Instruction	Count	Percent
Top by usage		
lwz gpr, <num> (<num>)	51434704	26.2
addic gpr, <num> , <num>	21115000	10.7
rlwinm gpr, <num> , <num> , <num> , <num>	20938000	10.6
add gpr, <num> , <num>	20938000	10.6
bc <num> , <num> , <num>	18302101	9.3
Top by memory access		
lwz gpr, <num> (<num>)	102869408	40.0
addic gpr, <num> , <num>	21115000	8.2
rlwinm gpr, <num> , <num> , <num> , <num>	20938000	8.1
add gpr, <num> , <num>	20938000	8.1
bc <num> , <num> , <num>	18302101	7.1
Top by delay		
lwz gpr, <num> (<num>)	51434704	26.2
addic gpr, <num> , <num>	21115000	10.7
rlwinm gpr, <num> , <num> , <num> , <num>	20938000	10.6
add gpr, <num> , <num>	20938000	10.6
bc <num> , <num> , <num>	18302101	9.3

Instruction	Count	Percent
Top by usage		
lwz gpr, <num> (<num>)	24442409	39.9
bc <num> , <num> , <num>	6797575	11.1
cmp <num> , <num> , gpr, <num>	6746195	11.0
addic gpr, <num> , <num>	4021073	6.5
rlwinm gpr, <num> , <num> , <num> , <num>	3844558	6.2
Top by memory access		
lwz gpr, <num> (<num>)	48884818	54.7
stw gpr, <num> (<num>)	7334084	8.2
bc <num> , <num> , <num>	6797575	7.6
cmp <num> , <num> , gpr, <num>	6746195	7.5
addic gpr, <num> , <num>	4021073	4.5
Top by delay		
lwz gpr, <num> (<num>)	24442409	39.9
bc <num> , <num> , <num>	6797575	11.1
cmp <num> , <num> , gpr, <num>	6746195	11.0
addic gpr, <num> , <num>	4021073	6.5
rlwinm gpr, <num> , <num> , <num> , <num>	3844558	6.2

Obrázek 9.1: Výstup profileru pro programy Bitcount, Dijkstrův algoritmus a Quicksort

Kapitola 10

Závěr

V jazyce CodAL jsem úspěšně namodeloval procesor PowerPC na úrovni instrukční sady a dle svého nejlepšího svědomí ho odladil a optimalizoval. K ověření správnosti implementace jsem vytvořil skript v jazyce Python pro automatizovanou kontrolu výsledků simulace. Jako testovací sadu jsem použil balíček 700 různě složitých programů v jazyce ANSI C, které mi byly poskytnuty výzkumnou skupinou Lissom. Tato testovací sada verifikovala přibližně 70% instrukční sady, funkčnost zbývajících instrukcí jsem ověřil jednotlivými assemblerovskými příkazy.

Během práce jsem pomohl odhalit nové chyby v generovaných nástrojích, jako je špatné načtení operandů typu byte a short z paměti big-endian architektur, či nesprávné výsledky logických operací na jednobitových operandech. Všechny tyto chyby jsem nahlásil přes Bugzillu[2], a tím pomohl zlepšit kvalitu vývojového prostředí Codasip Framework.

Výkonnost generovaného simulátoru z mého modelu jsem srovnal s konkurenčním simulátorem PSIM, který je součástí GNU PowerPC Debuggeru a je pravidelně aktualizován. Díky tomu je nejlepším možným oponentem pro vzájemné srovnání, neboť nám výsledky poskytují objektivní pohled na kvalitu generovaných nástrojů ve srovnání se současnou, vysoce optimalizovanou konkurencí. Jako srovnávací sadu jsem po konzultaci s vedoucím práce vybral tři programy: Bitcount (10 000 opakování), Dijkstrův algoritmus (1 000 opakování) a řadící algoritmus Quicksort (1 000 opakování). Výsledná hodnota doby a frekvence simulace je průměrem pětice měření. Očekával jsem, že generovaný simulátor bude pomalejší než jeho oponent, a to se potvrdilo. V porovnání s ním je výkonnost přibližně 55%.

Nejjednodušší procesor PowerPC 601 je taktován na 50MHz, to je přibližně čtyřnásobná frekvence, než se kterou dokážeme pracovat. Simulovaný procesor tedy není schopen pracovat v reálném čase.

Simulace s profilerem ukázala zajímavé informace o vnitřních dějích probíhajících v simulátoru. Překvapilo mě, že ani jeden z programů nebyl tvořen více než třinácti různými instrukcemi. Nejčastěji se vyskytující instrukcí napříč testovanými programy bylo lwz (načtení slova z paměti). Takové zjištění jsem očekával, neboť koresponduje s rysy RISC procesorů. S jejich pamětí nelze provádět jiné operace než LOAD/STORE, výpočty pak probíhají nad rychlejšími registry.

V případě potřeby je snadno možné domodelovat jednotku pro práci s čísly v plovoucí řádové čárce. Pokud bude výzkumná skupina Lissom využívat můj model pro další účely, lze ho zkvalitnit specifikací taktování a vytvořit tak model na úrovni cyklů, který procesor popisuje věrněji.

Literatura

[1] Apple Computer: Programmer's Introduction to PowerPC [online]. <http://physinfo.ulb.ac.be/divers_html/powerpc_programming_info/intro_to_ppc/ppc0_index.html>, 1995 [cit. 2013-02-21].

[2] Bugzilla: Systém pro hlášení chyb. <https://codasip.com/issue/report>.

[3] Freescale Semiconductor: Programming Environments Manual for 32-Bit Implementations of the PowerPC Architecture, Rev. 3 [online]. <http://www.freescale.com/files/product/doc/MPCFPE32B.pdf>, 2005 [cit. 2013-05-08].

[4] Karsh, B.: Understanding the PowerPC Architecture [online]. <http://www.mactech.com/articles/mactech/Vol.10/10.08/PowerPcArchitecture/index.html>, 1994 [cit. 2013-05-08].

[5] Kotásek, Z.: Materiály k předmětu ITP [online]. <https://www.fit.vutbr.cz/study/courses/ITP/public/>, 2013 [cit. 2013-05-08].

[6] Lissom: CodAL Manual, Interní dokumentace projektu. 2012.

[7] Lissom: Codasip Framework Manual, Interní dokumentace projektu. 2012.

[8] Masařík, K.: Jazyky pro popis architektur, Materiály k předmětu IPP. 2013 [cit. 2013-05-08].

[9] Olivka, P.: Studijní materiál pro předmět Architektury počítačů [online]. <http://poli.cs.vsb.cz/edu/arp/down/procrisc.pdf>, 2010 [cit. 2013-05-08].

[10] Schwarz, J.: Materiály k předmětu IMP [online]. <https://wis.fit.vutbr.cz/FIT/st/course-files-st.php/course/IMP-IT/lectures>, 2013 [cit. 2013-05-08].

[11] Wikipedia: Vestavěný systém [online]. <http://cs.wikipedia.org/wiki/Vestav%C4%9Bn%C3%BD_syst%C3%A9m>, 2013 [cit. 2013-05-08].

Příloha A

Obsah CD

Přiložené medium obsahuje následující adresářovou strukturu.

```
├── Model
│   └── src
├── Thesis
│   ├── src
│   └── thesis.pdf
├── Manual
│   └── manual.txt
├── Testy
│   ├── lib
│   ├── src
│   ├── crt0.s
│   └── run_testsuite.py
└── Seznam instrukcí
    └── instruction_set.html
```

- Model — Model procesoru PowerPC
- Thesis — Bakalářská práce
- Manual — Návod sestavení modelu
- Testy — Kompletní testovací sada
- Seznam instrukcí — Instrukční sada